

Computer Architecture – Project Report

*Sergio Rico (curious.matter@gmail.com), Ertong Zhang (zhangertong@gmail.com),
Vlad Chiriacescu (vlad_chiriacescu@yahoo.com), ZhongYin Zhang (zzy4032@gmail.com)*

Completion date: May 05, 2011

1. Abstract

In this report we present a design solution for a pipelined processor, having as an instruction set a subset of the known RISC set, a static branch predictor, forwarding mechanism, method for avoiding control hazards, instruction and data caches. The algorithm performed by the designed processor is the well-known bubble sort algorithm.

An important issue we dealt with was the choice of an appropriate instruction set architecture. We chose a small subset of the RISC set (5 instructions) and implemented a sixth one, called UART for communication with the UART port on the FPGA board used. For the static branch predictor we used an always not taken method since it is simple and as compared to always taken method, we don't have to compute the address of the instruction to jump at. For the forwarding mechanism, we obtain the data from the EXE, DM and WB stages to the ID stage of the next instruction and then decide in this stage what type of forwarding needs to be done according to what data dependencies are observed. Regarding the branch predictor used, we implemented the always not taken method. When the branch is taken, we introduce a bubble in the pipeline. The data cache used is 2 KB direct-mapped one that uses a write-back technique. This was observed to be very efficient since the cache is rather large for the given problem and the number of write-backs is rather small.

The organization of this report is as follows. The next section of the report is a brief theoretical overview that backs up our motivation for realizing this project. In section 3, we specify the features and functional details regarding our pipeline design, forward mechanism, control hazard prevention method and implemented cache.

In section 4, we present the actual results we obtained using four different test cases. As can be seen from this section, our processor successfully executes the bubble sort algorithm on every test case (set of unsorted numbers) presented. For a more visual understanding of these results, screenshots with sorted data are displayed in this section (numbers are presented in complement 1 format). These screenshots were taken from the Putty terminal that was used to display the results on the computer. Apart from displaying the sorted array for each test case, we also output a number of performance measures: Total Branch Count, Total Miss predicted Branches, Instruction Memory Access Count, Instruction Cache Misses, Data Memory Access Count, Data Cache Misses and Data Cache Write-Backs.

Conclusions and future work proposals are given in section 5.

2. Background Introduction

Motivation

One of the main driving force that made us successfully realize this project was getting some good hands-on general experience with processor design and its related technical issues. Our team members have different backgrounds and some of us are more related to software engineering or artificial intelligence fields.

But in an accelerated technical landscape where hardware and software are more and more interleaved, ideas from software, AI and also from theoretical Computer Science can fully impact hardware design of 21st century machines. Therefore, we consider that interaction between hardware related and applications/theory related people in such a hardware design project is of a great relevance.

More specifically, in a computer architecture era of an ever increasing gap between processor and memory performance, multidisciplinary approaches can lead to extraordinary solutions to this issue and leverage the processor power of today's and tomorrow's technical systems.

RISC Instruction Set Architecture

Reduced instruction set computing, or RISC is a CPU design strategy based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A computer based on this strategy is called a RISC computer. Most RISC architectures have three classes of instructions:

a) ALU instruction

These instructions take either two registers or a register and a sign-extended immediate, operate on them and store the result into a third register. Typical operations include add, subtract and logical operations (such as AND and OR).

b) Load and store instructions

In the case of a load instruction, the instruction can read an operand from the memory. In the case of a store instruction, the instruction can write an operand to memory.

c) Branches and jumps

Branches are conditional transfers of control, while jumps are unconditional transfers of control.

Single Cycle Processor Data Path

The cycle types of a single cycle processor are as follows:

- a) Instruction fetch cycle (IF)** - In this cycle the program counter (PC) is sent to memory and the current instruction is fetched from memory.
- b) Instruction decode/register fetch cycle (ID)** - Instruction is decoded and registers corresponding to register source are read. While they are read, an equality test is performed on the registers, for a possible branch.
- c) Execution/effective address cycle (EX)** - The ALU operates on the operands prepared in the prior cycle.
- d) Memory access (MEM)** - If the instruction is a load, memory does a read using the effective address computed in the execution cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.
- e) Write-back cycle (WB)** - Writes the result into the register file, whether it comes from the memory system (for a load instruction) or from the ALU (for an ALU instruction).

Pipelines and hazards

Pipelining increases the CPU instruction throughput – the number of instructions completed per unit of time – but it does not reduce the execution time of an individual instruction. In fact, it usually, slightly increases the execution time of each instruction due to overhead in the control of the pipeline. The increase in instruction throughput means that a program runs faster and has lower total execution time, even though no single instruction runs faster.

In our project, the 5 clock cycle executions presented above are pipelined by simply starting a new instruction on each clock cycle. Each of the clock cycles from the previous section becomes a pipe stage – a cycle in the pipeline. This results in the execution pattern shown below:

Instruction number	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

There are situations, called **hazards**, that prevent the next instruction in the instruction stream from executing during its designated clock cycle, thus reducing performance from the ideal speedup gained by pipelining.

There are three classes of hazards:

- a) **Structural hazards** - Those arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
- b) **Data hazards** - This type of hazard appears when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- c) **Control hazards** - Those arise from the pipelining of branches and other instructions that change the program counter (PC).

Branch predictors

In computer architecture, a **branch predictor** is a digital circuit that tries to guess which way a branch (e.g. an if-then-else structure) will go before this is known for sure. The purpose of the branch predictor is to improve the flow in the instruction pipeline.

Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of the executed code. Instead it predicts the outcome of a branch based solely on the branch instruction.

In the present work we implemented a static branch predictor that predicts the branch as taken. More details about its functioning are offered in the design section.

Instruction and Data Caches

Memory cache is the fastest type of memory after the CPU registers, thus the cache design and caching strategies directly impact overall system performance.

In the present work there are two types of caches included in our product. **Instruction cache** is to speed up executable instruction fetch. **Data cache** is to speed up data fetch and store.

Caching data that is only read is easy since the copy in the cache and memory will be identical. Caching writes is more complicated since data in cache and main memory needs to be kept consistent. In order to achieve this, one of two main strategies can be used. A write-through cache updates the item in the cache and writes through to update main memory. A write-back cache only updates the copy in the cache. When the block is about to be replaced, it is copied back to memory.

For our processor we employed a write-back cache since it is more efficient, especially when blocks in the cache are not frequently replaced.

3. Design

Instruction Set Architecture

We used the following instructions as the Instruction Set Architecture:

Add immediate (addi)

Load word (lw)

Set on less than (signed) (slt)

Branch on equal (beq)

Store word (sw)

UART (for communication with the UART RS232 transceiver on the Altera board)

Processor inputs and outputs

The processor has 4 inputs and 1 output:

Inputs

Reset: The reset signal is mapped to KEY0 button of the board. The signal has the value of 1 (high) when button is not pressed. When the button is pressed, reset signal becomes 0 (low). Clock: The CPU works at the maximum clock frequency on the board (50Mhz) or slower.

The input program is the bubble sort algorithm and the input data is an array of complement 1 signed 32bit integers. The array's maximum size is 128.

Outputs

The CPU result is printed out via the RS232 and it consists of the sorted array of numbers from lowest to highest value in hexadecimal. Each number has its own line on the terminal screen.

It then prints out 7 numbers that correspond to:

- 1) Total Branch Count
- 2) Total Miss predicted Branches
- 3) Instruction Memory Access Count
- 4) Instruction Cache Misses
- 5) Data Memory Access Count
- 6) Data Cache Misses
- 7) Data Cache Write-Backs

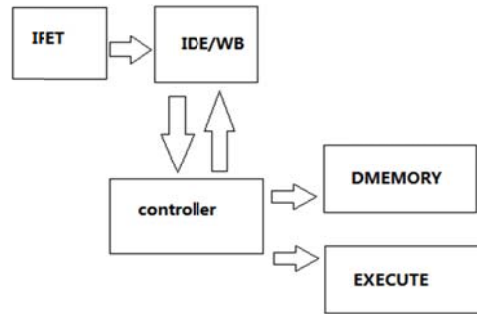
VHDL Design overview

In order to realize the pipeline we implemented in VHDL latches between the five stages involved: Instruction Fetch, Instruction Decode, Execute, Memory Access and Write Back. The branch predictor uses the branch taken method. For caching, we employed the write-back technique. Results are transmitted to the connected computer via the RS232 port and displayed in a Putty terminal.

1. Single cycle CPU

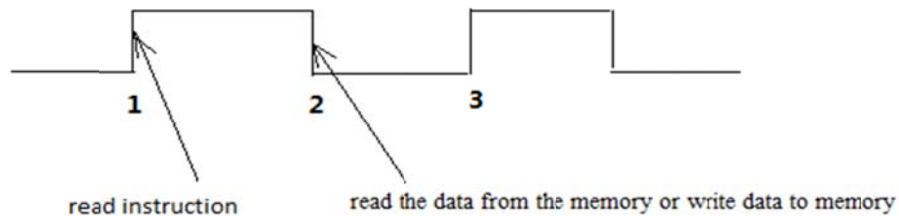
Functional interaction

The functional interaction inside the single cycle processor can be summarized with the following diagram:



In the IFetch (IFET) stage data is obtained from memory and then is passed into the IDecode/Write Back (IDE/WB) stage in order to get the instruction type. The controller controls the IDE/WB and decides when to write to the registers, when to load from memory or store to it and directs the actions in the EXECUTE stage when it gets the instruction type from IDE/WB.

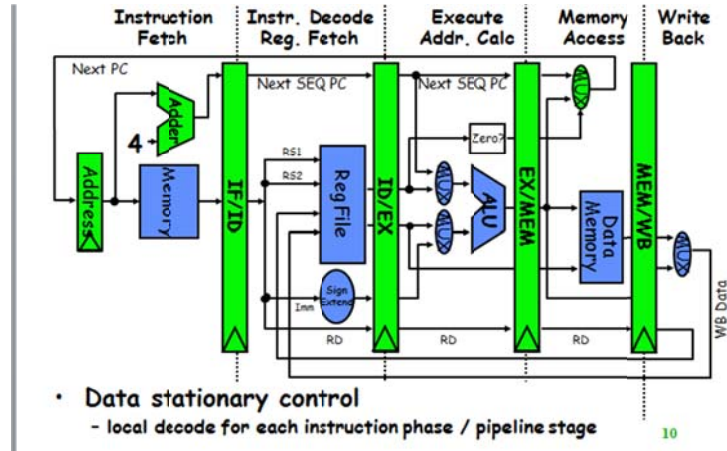
Clock usage



In the case of single cycle processor, an instruction is finished in one processor cycle.

1. On the rising edge of the clock, the instruction is read and the results written to the registers
2. On the falling edge of the clock, data is read from the memory or written to the memory
3. On the next rising edge of the clock, the next instruction is read and the process continues in the same manner.

2. Pipelined CPU

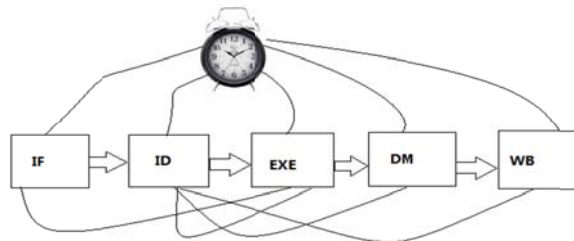


In each cycle, every stage should get or compute the data for the next stage. For example the IF and ID stage, ID stage needs the IR and PC from IF stage, so we should output the IR and PC in IF stage, and then in ID stage, the IR and PC are the input. After determining which are the inputs and outputs of each stage, we focused on determining the functional details of each stage.

One of the most important issues we encountered during the design of the pipelined processor is that we have stages when we have to deal with memory, which also needs a clock. If we use the rising edge and we put data read from memory in the process controlled by clock, the result is the last time's data. This happens because the current data is still being read. By putting this part out of the VHDL process, we can output the data as soon as it is read from memory.

3. Data forwarding mechanism

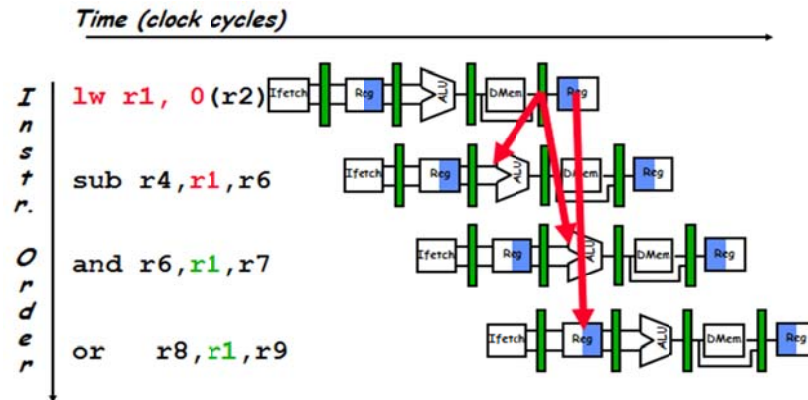
There are different methods to realize data forwarding in order avoid data hazards. The one we use is a simple one and it is implemented in the ID stage. Its description is presented below.



Before data is sent to the execution (EXE) stage, the ID stage asks the EXE, DM and WB stages to send their current data. Then in the ID stage, a dependency check is made on the data received from those other stages. The ID will change the data in the two registers A and B according to the dependencies found and avoid many of the possible data hazards.

4. Load instruction issue

LW instruction does not get the required data until this data is moved to the Data Memory (DM) part. With the data forwarding mechanism presented above, the best that can be achieved is to forward the EXE output of one instruction to the EXE input of the next instruction in case of data dependencies. But if we use the previous mechanism, we cannot solve the data dependency between a load (LW) instruction and the next instruction since we need the LW output at the EXE input of next instruction. This happens all in all situations except the case when the next instruction is a store instruction. The issue is displayed below using an example with load, and, subtract and or instructions:

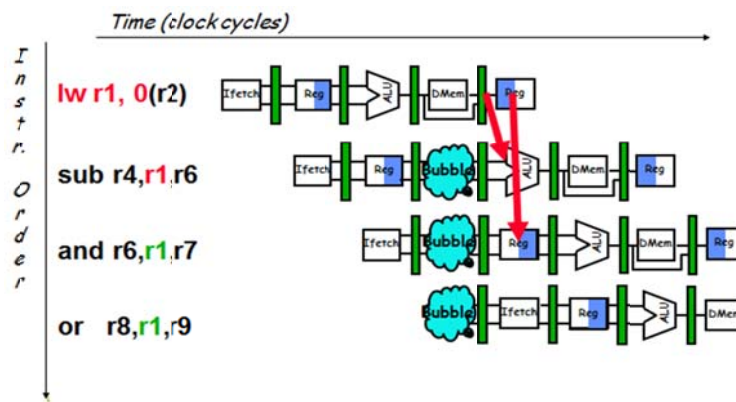


Our solution for this issue is simple and straight-forward: we introduce a bubble in the pipeline.

The detailed process is as follows:

1. We decode the instruction in IF stage, so we can know which instruction is LW (load) in IF stage.
2. If the instruction is a load instruction, we add a bubble, even though we do not know whether there is a hazard between this load instruction and the next instruction.
3. Therefore the pipeline is stalled for one time. As we are going to send a bubble, we do not need to read instruction next time.

The solution is displayed below using the same sequence of operations from previous example:



5. Branch prediction and control hazard prevention method

The inspiration for realizing control hazard prevention method came from how we dealt with the memory hazards. We employed the branch not taken method. If the branch is taken, we introduce a bubble in the pipeline and flush the existing data. We change every output data to be "000...". In essence, this doesn't affect the system functioning.

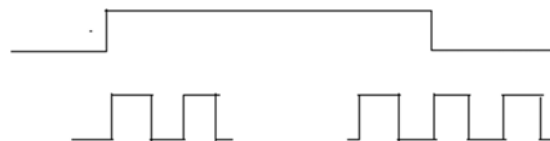
6. Cache

Parameter description

Block size	4*32B
Associativity	1
Number of blocks in cache	16
Total	2 KB

Time sequence

As we have to read the memory in case of cache miss, we use a two-tier time sequence.



We use the low frequency clock to control the pipeline and the high frequency one to fasten the transmission of data from memory to cache. Thus, we make sure we get the right data even if there is a conflict miss and the data is not in the cache.

Cache miss

If data is found in the cache then everything is ok and the fastness of direct mapped cache is fully exploited (less time to search for data in cache as compared to higher associativity).

The problem arises when the processor tries to read data from cache but the data is not found. In this case, we fetch the data. In order to do this, we take advantage that there are two edges in one clock cycle. We use the falling edge to check if it needs to read from memory or write to memory. If so, we read from or write to memory. In the rising edge, if we know from the falling edge part we should read or write, we will bring the data from memory to cache (in case of a read) or send the data to memory (in case of a write). After completing this process, we set the dirty and valid bits.

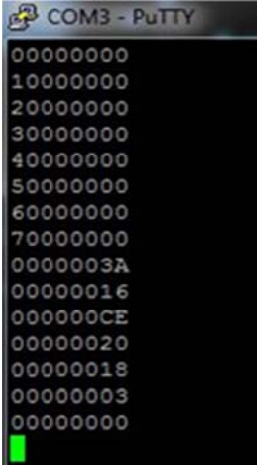
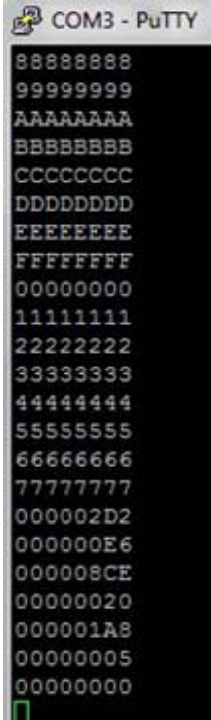
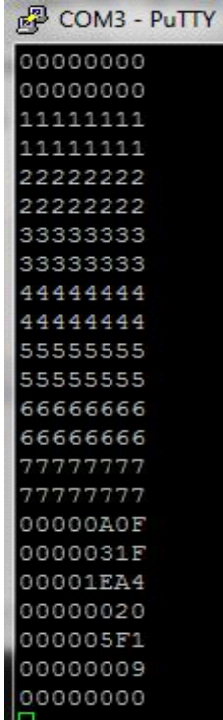
Write method

The write method used is write back. Since the cache is large (2KB) compared to data required for the given problem, there are not many write backs and this method clearly outperforms a write-through method.

4. Evaluation

In this section we present the results we obtained by sorting four array sets using our designed processor.

After displaying the sorted numbers in complement 1 format, we also output a number of performance measures: Total Branch Count, Total Miss predicted Branches, Instruction Memory Access Count, Instruction Cache Misses, Data Memory Access Count, Data Cache Misses and Data Cache Write-Backs. These are also displayed in complement 1 format.

Test case 1:	Test case 2:	Test case 3:
		

5. Conclusions and future work proposals

In this project we successfully designed a functional solution for a pipelined processor with static branch predictor, forwarding mechanism, control hazard prevention mechanism, instruction and data caches. We ran the bubble sort algorithm for 4 different test cases and the designed processor flawlessly sorted the arrays in all of the 4 test cases.

The work at this project enabled our team members to get a grasp on how important is the choice of an appropriate instruction set given the problem to be solved and also the hardware available (in our case an Altera board with RS232 and USB Blaster ports for bidirectional communication between the board and computer used for processor design). Furthermore, we had the unique chance of getting some good hands-on experience with VHDL design and related issues. Lastly, we obtained a more clear and technical

perspective over how forward mechanism, avoiding hazards methods and a good choice of cache features affect the overall system performance.

Regarding future work for this project we propose experimentation with different algorithms to be performed on the processor. This would help observing the obstacles that arise when trying to come up with an appropriate ISA for more algorithms.

When more complex algorithms are ran and the amount of data to be processed is substantial, the implementation of very powerful dynamic branch predictors such as neural branch predictors will definitely improve processor performance. Professor Lucian Vintan from University of Sibiu, Romania came up with the first neural branch predictor (1999) and over the years the whole idea of a neural branch predictor became more and more promising. Most of the state of the art branch predictors are using a perceptron predictor. Intel already implemented this idea in one of the IA-64's simulators (2003).

This branch predictor proposal is also in clear relation with our motivation section – interaction between hardware and mainly AI related fields. Much more powerful predictors have been developed in the field of neural networks and in general in the AI field than the perceptron predictor used right now in the best of the best branch predictors.

Regarding the improvement in cache performance, we propose the implementation of a simple pseudo-associative cache that keeps the speed advantage of a direct mapped cache but decreases conflict misses. This would clearly be advantageous when the amount of data is substantially higher and memory – cache size ratio higher than the one used in the current implementation.

References

1. John L. Hennessy and David A. Patterson, Computer Architecture - A Quantitative Approach, 4th Edition, Morgan Kaufmann Publishers Inc., 2007